

## IB CS1 Test2

Please write your code and upload to your own project page (create a new one called Test2\_solutions).

1.

Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the relational operators `<`, `>`, etc.

For example, `is_sorted([1,2,2])` should return `True` and `is_sorted(['b','a'])` should return `False`.

2.

Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams.

3.

The (so-called) Birthday Paradox:

1. Write a function called `has_duplicates` that takes a list and returns `True` if there is any element that appears more than once. It should not modify the original list.
2. If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the `randint` function in the `random` module.

You can read about this problem at [http://en.wikipedia.org/wiki/Birthday\\_paradox](http://en.wikipedia.org/wiki/Birthday_paradox), and

4.

Write a function called `remove_duplicates` that takes a list and returns a new list with only the unique elements from the original. Hint: they don't have to be in the same order.

5.

We can specify a transducer (a process that takes as input a sequence of values which serve as inputs to the state machine, and returns as output the set of outputs of the machine for each input) as a state machine (SM) by specifying:

- a set of *states*,  $S$ ,
- a set of *inputs*,  $I$ , also called the *input vocabulary*,
- a set of *outputs*,  $O$ , also called the *output vocabulary*,
- a *next-state function*,  $n(i_t, s_t) \mapsto s_{t+1}$ , that maps the input at time  $t$  and the state at time  $t$  to the state at time  $t + 1$ ,
- an *output function*,  $o(i_t, s_t) \mapsto o_t$ , that maps the input at time  $t$  and the state at time  $t$  to the output at time  $t$ ; and
- an *initial state*,  $s_0$ , which is the state at time  $0$ .

Here are a few state machines, to give you an idea of the kind of systems we are considering.

- A *tick-tock* machine that generates the sequence  $1, 0, 1, 0, \dots$  is a finite-state machine that ignores its input.
- The controller for a digital watch is a more complicated finite-state machine: it transduces a sequence of inputs (combination of button presses) into a sequence of outputs (combinations of segments illuminated in the display).
- The controller for a bank of elevators in a large office building: it transduces the current set of buttons being pressed and sensors in the elevators (for position, open doors, etc.) into commands to the elevators to move up or down, and open or close their doors.

The very simplest kind of state machine is a pure function: if the machine has no state, and the output function is purely a function of the input, for example,  $o_t = i_t + 1$ , then we have an immediate functional relationship between inputs and outputs on the same time step. Another simple class of SMs are *finite-state machines*, for which the set of possible states is finite. The elevator controller can be thought of as a finite-state machine, if elevators are modeled as being only at one floor or another (or possibly between floors); but if the controller models the exact position of the elevator (for the purpose of stopping smoothly at each floor, for example), then it will be most easily expressed using real numbers for the state (though any real instantiation of it can ultimately only have finite precision). A different, large class of SMs are describable as *linear, time-invariant* (LTI) systems. We will discuss these in detail chapter ??.

## Examples

Let's look at several examples of state machines, with complete formal definitions.

### Language acceptor

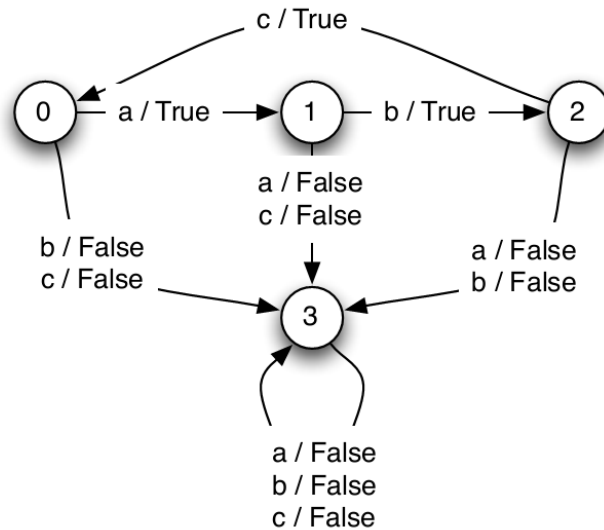
Here is a finite-state machine whose output is *true* if the input string adheres to a simple pattern, and *false* otherwise. In this case, the pattern has to be  $a, b, c, a, b, c, a, b, c, \dots$

It uses the states 0, 1, and 2 to stand for the situations in which it is expecting an  $a$ ,  $b$ , and  $c$ , respectively; and it uses state 3 for the situation in which it has seen an input that was not the one that was expected. Once the machine goes to state 3 (sometimes called a *rejecting state*), it never exits that state.

$$\begin{aligned} S &= \{0, 1, 2, 3\} \\ I &= \{a, b, c\} \\ O &= \{true, false\} \\ n(s, i) &= \begin{cases} 1 & \text{if } s = 0, i = a \\ 2 & \text{if } s = 1, i = b \\ 0 & \text{if } s = 2, i = c \\ 3 & \text{otherwise} \end{cases} \\ o(s, i) &= \begin{cases} false & \text{if } n(s, i) = 3 \\ true & \text{otherwise} \end{cases} \\ s_0 &= 0 \end{aligned}$$

**Figure 4.1** shows a state transition diagram for this state machine. Each circle represents a state. The arcs connecting the circles represent possible transitions the machine can make; the arcs are labeled with a pair  $i, o$ , which means that if the machine is in the state denoted by a circle with label  $s$ , and gets an input  $i$ , then the arc points to the next state,  $n(s, i)$  and the output generated  $o(s, i) = o$ . Some arcs have several labels, indicating that there are many different inputs that will cause the same transition. Arcs can only be traversed in the direction of the arrow.

For a state transition diagram to be complete, there must be an arrow emerging from each state for each possible input  $i$  (if the next state is the same for some inputs, then we draw the graph more compactly by using a single arrow with multiple labels, as you will see below).



**Figure 4.1** State transition diagram for language acceptor.

We will use tables like the following one to examine the evolution of a state machine:

| time   | 0     | 1     | 2     | ... |
|--------|-------|-------|-------|-----|
| input  | $i_0$ | $i_1$ | $i_2$ | ... |
| state  | $s_0$ | $s_1$ | $s_2$ | ... |
| output | $o_1$ | $o_2$ | $o_3$ | ... |

For each column in the table, given the current input value and state we can use the output function  $o$  to determine the output in that column; and we use the  $n$  function applied to that input and state value to determine the state in the next column. Thus, just knowing the input sequence and  $s_0$ , and the next-state and output functions of the machine will allow you to fill in the rest of the table.

For example, here is the state of the machine at the initial time point:

| time   | 0     | 1 | 2 | ... |
|--------|-------|---|---|-----|
| input  | $i_0$ |   |   | ... |
| state  | $s_0$ |   |   | ... |
| output |       |   |   | ... |

Using our knowledge of the next state function  $n$ , we have:

|        |       |       |   |     |
|--------|-------|-------|---|-----|
| time   | 0     | 1     | 2 | ... |
| input  | $i_0$ |       |   | ... |
| state  | $s_0$ | $s_1$ |   | ... |
| output |       |       |   | ... |

and using our knowledge of the output function  $o$ , we have at the next input value

|        |       |       |   |     |
|--------|-------|-------|---|-----|
| time   | 0     | 1     | 2 | ... |
| input  | $i_0$ | $i_1$ |   | ... |
| state  | $s_0$ | $s_1$ |   | ... |
| output | $o_1$ |       |   | ... |

This completes one cycle of the statement machine, and we can now repeat the process.

Here is a table showing what the language acceptor machine does with input sequence ('a', 'b', 'c', 'a', 'c', 'a', 'b'):

|        |      |      |      |      |       |       |       |   |
|--------|------|------|------|------|-------|-------|-------|---|
| time   | 0    | 1    | 2    | 3    | 4     | 5     | 6     | 7 |
| input  | 'a'  | 'b'  | 'c'  | 'a'  | 'c'   | 'a'   | 'b'   |   |
| state  | 0    | 1    | 2    | 0    | 1     | 3     | 3     | 3 |
| output | True | True | True | True | False | False | False |   |

The output sequence is (True, True, True, True, False, False, False).

Clearly we don't want to analyze a system by considering all input sequences, but this table helps us understand the state transitions of the system model.

*To learn more:* Finite-state machine language acceptors can be built for a class of patterns called *regular languages*. There are many more complex patterns (such as the set of strings with equal numbers of 1's and 0's) that cannot be recognized by finite-state machines, but can be recognized by a specialized kind of infinite-state machine called a *stack machine*. To learn more about these fundamental ideas in *computability theory*, start with the Wikipedia article on **Computability\_theory\_(computer\_science)**

Implement this state machine.

6.

### Accumulator

Here is a machine whose output is the sum of all the inputs it has ever seen.

$$S = \text{numbers}$$

$$I = \text{numbers}$$

$$O = \text{numbers}$$

$$n(s, i) = s + i$$

$$o(s, i) = n(s, i)$$

$$s_0 = 0$$

Here is a table showing what the accumulator does with input sequence 100, -3, 4, -123, 10:

| time   | 0   | 1   | 2   | 3    | 4   | 5   |
|--------|-----|-----|-----|------|-----|-----|
| input  | 100 | -3  | 4   | -123 | 10  |     |
| state  | 0   | 100 | 97  | 101  | -22 | -12 |
| output | 100 | 97  | 101 | -22  | -12 |     |

Implement this state machine.